OCR Computer Science A Level

2.3.1 Algorithms for the Main Data Structures

Advanced Notes

Specification:

- Stacks
- Queues
- Linked lists
- Trees
- Traversal of trees
    - Depth-first (post-order)
    - Breadth-first

Algorithms for the Main Data Structures

Algorithms and data structures go hand in hand. Each data structure has its own algorithms associated with it, allowing the data to be manipulated in useful ways.

All of the data structures mentioned in these notes are covered in greater detail in the notes for 1.4.2 Data Structures.

Stacks

Stacks are an example of a first in, last out (FILO) data structure. They are often implemented as an array and use a single pointer which keeps track of the top of the stack (called the top pointer). This points to the element which is currently at the top of the stack.

The top pointer is initialised at -1; this is because the first element in the stack is in position 0, and having the top initialised at 0 would suggest there is an element in the stack, when in fact the stack is empty.

Algorithms for stacks include adding to the stack, removing from the stack and checking whether the stack is empty/full. These have their own special names, as shown in the table below.

Operation	Name
Check size	`size()`
Check if empty	`isEmpty()`
Return top element (but don't remove)	`peek()`
Add to the stack	`push(element)`
Remove top element from the stack and return removed element	`pop()`

size()

Size returns the number of elements on the stack. The pseudocode is as simple as returning the value of the top pointer plus one (remember that the first element is in position 0).

```
size()
    return top + 1
```

isEmpty()

To check whether a stack is empty, we need to check whether the top pointer is less than 0. If it is, then the stack is empty, otherwise there is data in the stack

```
isEmpty()
    if top < 0:
        return True
    else:
        return False
    endif
```

peek()

To return the item at the top of the stack, without removing it, simply return the item at the position indicated by the top pointer. For these examples, we'll assume our stack is an array called A.

Don't forget to check that the stack has data in it before attempting to return data though, as an empty stack could cause errors. It's useful to use the `isEmpty` function here.

```
peek()
    if isEmpty():
        return error
    else:
        return A[top]
    endif
```

push(element)

To add an item to a stack, the new item must be passed as a parameter. Firstly, the top pointer is updated accordingly. Then the new element can be inserted at the position of the top pointer.

```
push(element)
     top += 1
     A[top] = element
```

pop()

To remove an item from a stack, the element at the position of the top pointer is recorded before being removed, and then the top pointer decremented by one before the removed item is returned. As with `peek()`, it's important to check that the stack isn't empty before attempting a pop.

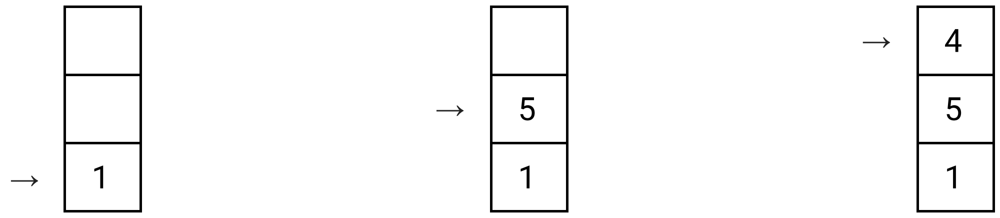
```
pop()
     if isEmpty():
          return error
     else:
          toRemove = A[top]
          A[top] = “”
          top -= 1
          return toRemove
     endif
```

Example

What would be the result of the following operations on a 3-element stack?

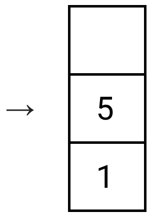
```
push(1)
push(5)
push(4)
peek()
pop()
isEmpty()
push(2)
push(3)
pop()
pop()
```

The first three operations `push` the items 1, 5 and 4 to the stack in that order.

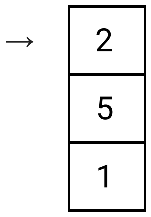


The next operation is a `peek`. This returns the item at the top of the stack, but doesn’t change the appearance of the stack. Therefore this operation returns **4** and the stack remains the same.

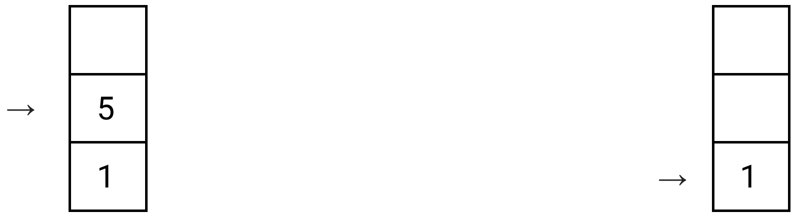
Next is `pop`. This removes the item at the top of the stack, **4**, and returns it before the top pointer moves down one place.



Now `isEmpty` is carried out. The stack is not empty and so **False** is returned. Next 2 is pushed onto the stack.



Now 3 is pushed onto the stack, but as the stack is full, an **error** is returned and the stack stays the same. Now two consecutive `pop`s are carried out, removing 2 and 5 in that order and outputting the values **2, 5**.



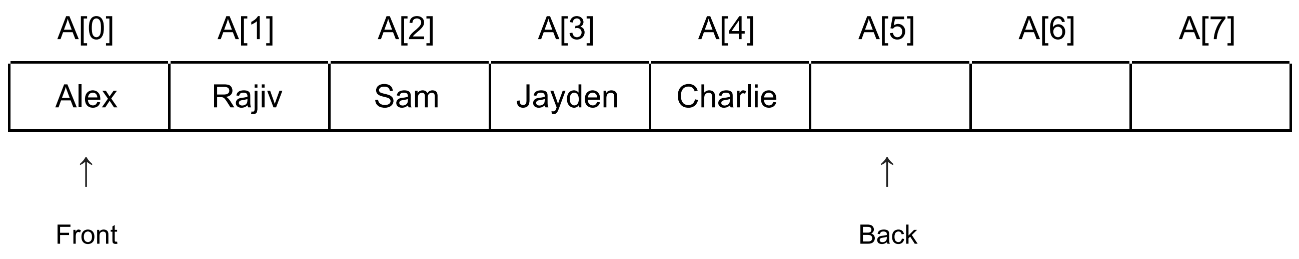
The final state of the stack is shown to the left above. The output from the operations is:

`4, 4, False, 2, 5`

Queues

Queues are a type of first in, first out (FIFO) data structure. Just like stacks, queues are often represented as arrays. However, unlike stacks, queues make use of two pointers: front and back. While front holds the position of the first element, back stores the next available space.

Operations which can be carried out on queues are similar to those associated with stacks, but be aware - some have different names.



Operation	Name
Check size	`size()`
Check if empty	`isEmpty()`
Return front element (but don't remove)	`peek()`
Add to the queue	`enqueue(element)`
Remove front element from the queue and return removed element	`dequeue()`

size()

To work out the size of a queue, simply subtract the value of front from back. If front is at 0 and back is at 5, there are 5 elements in the queue.

```
size()
     return back - front
```

isEmpty()

When a queue is empty, front and back point to the same position. To check whether a queue is empty, just check whether the two pointers hold the same value.

```
isEmpty()
     if front == back:
          return True
     else:
          return False
     endif
```

peek()

Just as with a stack, peek returns the element at the front of the queue without removing it.

```
peek()
     return A[front]
```

enqueue(element)

To add an element to a queue, the element is placed in the position of the back pointer and then back is incremented by one.

```
enqueue(element)
     A[back] = element
     back += 1
```

dequeue()

Items are removed from a queue from the position of the front pointer. Just as with stacks, it's important to check that the queue isn't empty before trying to dequeue an element.

```
dequeue()
     if isEmpty():
          return error
     else:
          toDequeue = A[front]
          A[front] = “”
          front += 1
          return toDequeue
```

Example

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
Alex	Rajiv	Sam	Jayden	Charlie			
↑ Front					↑ Back		

What would be the result of the following operations on the queue above?

```
dequeue()
enqueue("Julia")
size()
peek()
size()
dequeue()
isEmpty()
```

The first operation is `dequeue`, which removes Alex from the front of the queue and moves the front pointer to Rajiv. **Alex** is returned.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
	Rajiv	Sam	Jayden	Charlie			
	↑ Front				↑ Back		

Next, Julia is enqueued. The name is added at the position of the back pointer and the back pointer is moved to position 6.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
	Rajiv	Sam	Jayden	Charlie	Julia		
	↑ Front					↑ Back	

Now we check the size of the queue. `6-1  =` 5 and so **5** is returned. The next operation is `peek` which returns the item at the front of the queue, **Rajiv**, but does not change the queue otherwise.
The next operation is `size`, and because the queue hasn't changed as a result of the peek operation, **5** is returned again.

Next a `dequeue` is performed. Rajiv is returned, removed from the queue and the front pointer moved to Sam.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
		Sam	Jayden	Charlie	Julia		
		↑				↑	
		Front				Back	

Finally, the `isEmpty` command is performed. Because the values of front and back are not the same, False is returned.

The output of the operations is therefore:

Alex, 5, Rajiv, 5, Rajiv, False

Linked Lists

A linked list is composed of nodes, each of which has a pointer to the next item in the list. If a node is referred to as N, the next node can be accessed using N.next. The first item in a list is referred to as the head and the last as the tail.

Searching a list is performed using a linear search, carried out by sequential next operations until the desired element is found.

Trees

Trees are formed from nodes and edges, which cannot contain cycles and aren't directed. Trees are useful as a data structure because they can be traversed.

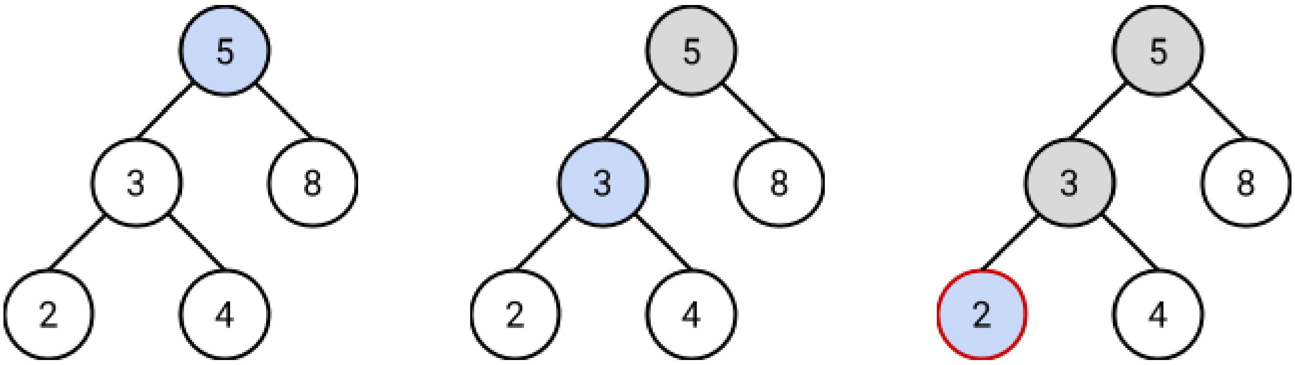
There are two types of traversal to cover: depth first (post-order) and breadth first. Both of these traversals can be implemented recursively and differ only in the order in which nodes are visited.

Depth first (post-order) traversal

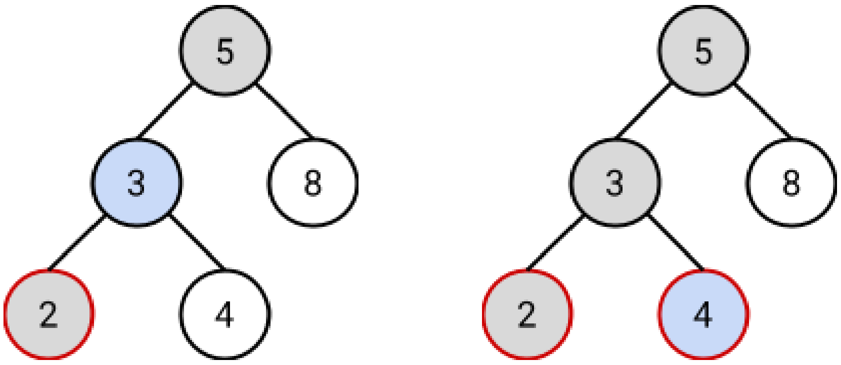
Depth first search goes as far down into the tree as possible before backtracking. The algorithm uses a stack and goes to the left child node of the current node when it can. If there is no left child then the algorithm goes to the right child.

If there are no child nodes, the algorithm visits the current node, outputting the value of the node before backtracking to the next node on the stack and moving right.

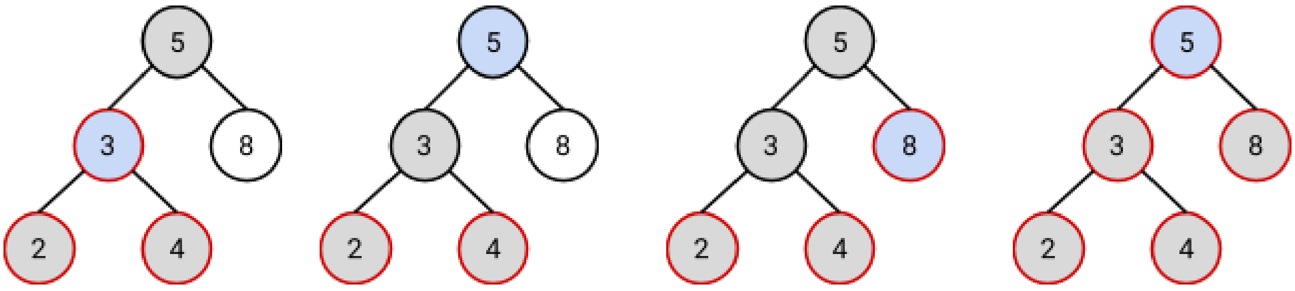
On the tree below, with depth first search starting at the node labelled 5, the algorithm moves left while possible, passing 3 before reaching 2. At this point, the algorithm can no longer move left and therefore 'visits' 2, outputting the value.



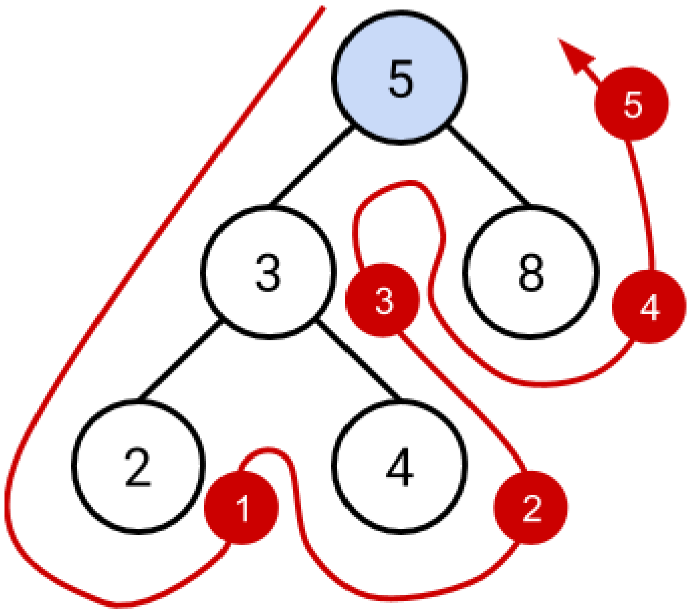
The algorithm now backtracks to 3 and moves right to 4, at which point it can progress no further so outputs the value and backtracks again.



The algorithm backtracks from 4 to 3, which it outputs (as all of its children have been visited) and continues to 5, where it can move right to 8. It outputs 8, backtracks to 5 and outputs 5 before terminating. The order is therefore **2, 4, 3, 8, 5**.



It can be easier to think of tree traversals as drawing a line around the tree, outputting as the line passes the node at a particular point. For depth first, nodes are output as they are passed on the right, like so:



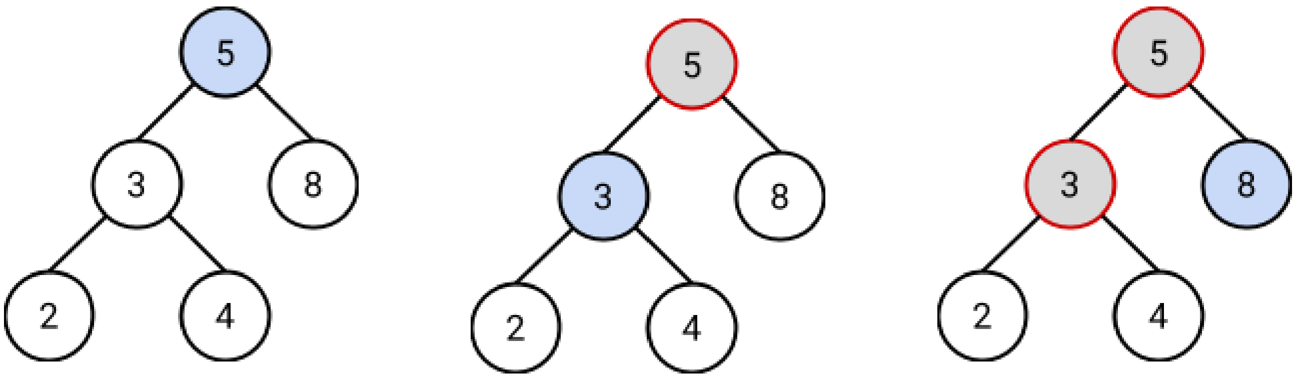
This gives the same result, **2, 4, 3, 8, 5**.

Other tree traversals, pre-order and in-order aren't required in the exam, but they can be performed using the method above, outputting the values as the line passes to the left and directly under the nodes respectively.

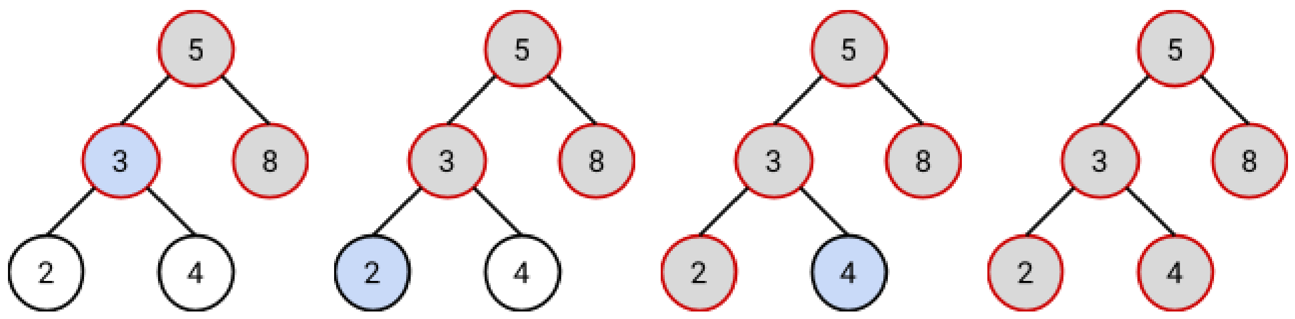
Breadth first

Starting from the left, breadth-first visits all children of the start node. The algorithm then visits all nodes directly connected to each of those nodes in turn, continuing until every node has been visited. Unlike depth first traversal (which uses a stack), breadth first uses a queue.

Starting at 5, which is visited immediately, the algorithm visits 3 and then 8.



As 8 has no children, the algorithm backtracks to 3 and visits its children, starting from the left. 2 and 4 are visited and the algorithm backtracks to 2, 3 and 5 before terminating.


5
3
8
2
4
5
3
8
2
4
5
3
8
2
4
5
3
8
2
4